Dinamičko programiranje

Primer 1:

Neka je data kvadratna matrica *A* realnih brojeva. Sa svakog polja je dozvoljeno preći samo na polje ispod ili na polje desno od tog polja. Potrebno je izabrati put od gornjeg levog polja do donjeg desnog polja, tako da zbir brojeva u poljima preko kojih se ide, bude maksimalan.

Rešenje:

Često je prva ideja koja pada na pamet isprobavanje svih mogućih puteva (što se lako realizuje rekurzivno) i pamćenje onog puta koji trenutno ima najveći zbir. Ova ideja nije dobra jer broj mogućih puteva raste veoma brzo. Tačnije, broj puteva za matricu od n+1 redova i kolona je $\binom{2n}{n} = \frac{(2n)!}{n! \cdot n!} \geq \frac{4^n}{2\sqrt{n}}$, dakle raste eksponencijanom brzinom sa porastom veličine matrice. Zbog ovako brzog rasta broja putanja, za nešto veće matrice, na primer 100 x 100, postupak je potpuno neupotrebljiv.

Da bismo lakše analizirali problem, uvedimo najpre nekoliko oznaka, i to:

- *A*(i, j) za broj koji se nalazi u polju (i, j);
- *M*(i, j) za maksimalni zbir na putu od polja (1, 1) do polja (i, j);
- *P*(i, j) za optimalan put od polja (1, 1) do polja (i, j) (tj. onaj sa zbirom *M*(i,j)); - *Q*(i, j) za problem nalaženja zbira *M*(i, j) i puta *P*(i, j).

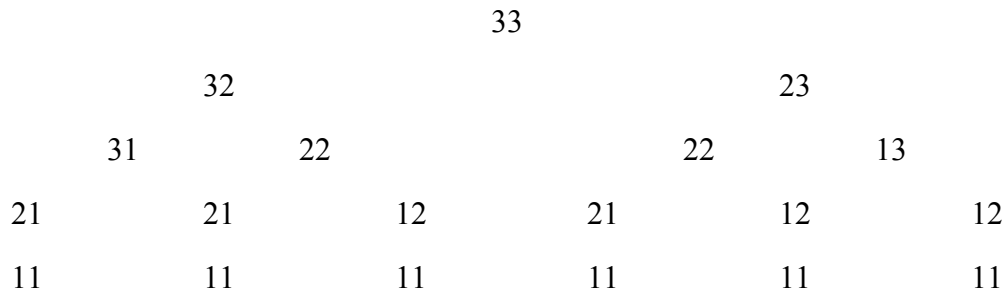
Primetimo da na polje (n, n) može da se dođe samo sa polja (n, n-1) ili (n-1, n), to jest, svaka putanja do polja (n, n) vodi preko jednog od dva pomenuta polja. Ako već znamo maksimalne zbirove *M*(n, n-1), *M*(n-1, n) do ovih polja, onda zbir *M*(n, n) možemo izračunati kao: *M*(n,n) = max(*M*(n, n-1), *M*(n-1, n)) + *A*(n,n)

To znači da je za rešavanje problema *Q*(n,n) dovoljno rešiti *Q*(n,n-1) i *Q*(n-1,n). Put *P*(n,n) se dobija dodavanjem polja (n, n) na onaj od puteva *P*(n-1, n) ili *P*(n, n-1) koji daje veći zbir *M*.

Prema tome, rešavanje polaznog problema se svodi na rešavanje dva podproblema potpuno istog tipa, ali manje veličine, plus jednostavno kombinovanje rešenja podproblema - poređenje dva zbira i dodavanje vrednosti završnog polja većem zbiru.

Za rešavanje podproblema nije dobro koristiti rekurziju, jer time bismo ponovo generisali sve puteve kroz matricu. Razlika je jedino u tome što na ovaj način razgranavamo od kraja putanje, umesto od početka. Eksponencijalni rast broja podproblema pri rekurzivnom rešavanju dolazi otuda što se većina podproblema javlja više puta. Na primer (kao što je već rečeno), radi rešavanja *Q*(n,n) rešićemo *Q*(n,n-1) i *Q*(n-1,n), a pri rešavanju svakog od ova dva podproblema, pojaviće se *Q*(n-1, n-1). Tako bi podproblem *Q*(n-1, n-1) bio rešavan dva puta, zajedno sa celim

stablom njegovih podproblema. Na primer, za n = 3 celo stablo rekurzivnih poziva bi izgledalo ovako:



Vidi se da se već problem $Q(2,2)$ (i celo stablo njegovih podproblema) javlja dva puta, a $Q(1,1)$ čak šest puta. Ukupno se kreira i rešava 19 problema umesto samo 9 koliko ih ima različitih. Za veće n gubici su, naravno, daleko više izraženi.

Da bi rešenje bilo efikasno, potrebno je svaki podproblem rešiti samo jednom. To se može obezbediti time što bismo podprobleme rešavali redom od najjednostavnijih prema složenijim i pri tome pamtili dobijena rešenja. Na sreću, ukupan broj podproblema je samo n·n, tako da ih relativno brzo možemo redom rešiti sve. Algoritam napisan po ovoj ideji izgledao bi ovako:

```
input(n)
for i = 1 to n
        for j = 1 to n
                input(a[i][j])
m[1][1] = a[1][1]
for i = 2 to n
        m[1][i] = a[1][i]+m[1][i-1]
        m[i][1] = a[i][1]+m[i-1][1]
for i = 2 to n
        for j = 2 to n
                if m[i][j-1] < m[i-1][j]
                        m[i][j] = a[i][j] + m[i-1][j]
                else
                        m[i][j] = a[i][j] + m[i][j-1]
output('Maksimalni zbir je ',m[n][n],' a postize se ovako:') WriteSolution(n,n);
```

gde je procedura za ispis rešenja:

```
WriteSolution(a,b)
if (a>1) and (b>1)
                if m[a-1,b] > m[a,b-1]
                        WriteSolution(a-1, b)
                        output('dole ')
```

```
        else
                WriteSolution(a, b-1)
                output ('desno ')
else if a==1
        for i = 2 to b output('desno ')
else
        for j = 2 to a output('dole ');
```

Koje su ključne osobine ovog problema, zahvaljujući kojima on može (i treba) da se rešava dinamičkim programiranjem?

Prvo, optimalno rešenje problema sadrži u sebi optimalna rešenja svojih podproblema. To znači da, ako se na optimalnom putu P(x,y) do polja (x,y) prelazi preko polja (u,v), tada je deo puta P(x,y) do polja (u,v) upravo optimalan put P(u,v) za odgovarajući podproblem Q(u,v). Još se kaže da problem ima optimalnu podstrukturu. Optimalnost podstrukture se lako dokazuje svođenjem na kontradikciju: pretpostavi se da podproblem Q(u,v) ima bolje rešenje i to se dovede u kontradikciju sa optimalnošću rešenja polaznog problema Q(x,y). Pri nalaženju maksimalnih zbirova i odgovarajućih optimalnih puteva koristilli smo optimalne puteve za podprobleme, to jest, oslanjali smo se na optimalnost podstrukture problema. Drugim rečima, znali smo da veliki deo rešenja većeg problema već imamo u rešenju nekog manjeg problema, samo je trebalo ustanoviti rešenje kojeg podproblema treba iskoristiti, i kako ga dopuniti do rešenja većeg problema.

Drugo, ukupan broj podproblema koji uopšte pojavljuju je relativno mali, tipično polinom od ulazne veličine (ovde je n^2). U isto vreme, rekurzivnim algoritmom se mnogi podproblemi kreiraju više puta, i svaki put se iznova rešavaju. Za probleme sa ovom osobinom kažemo da imaju preklapajuće podprobleme. Efikasnost dinamičkog programiranja dolazi do izražaja baš na ovakvim problemima, jer se svaki podproblem kreira i rešava samo jednom, a onda se porede i kombinuju rešenja podproblema da bi se rešili problemi višeg reda, sve do polaznog (glavnog) problema.

Primer 2:

Perica može da se penje uz stepenice po 1, 2, … do d stepenica u jednom koraku. Na koliko načina Perica može da se popne uz stepenište od n stepenika?

Rešenje:

Označimo sa B(m) broj načina da se pređe m stepenika u koracima dužine do d. Ako je m = 0, postoji samo jedan način da Perica dovrši penjanje, a to je da ne radi više ništa jer se već nalazi na cilju. Prema tome, B(0) = 1.

Neka je sada m > 0 i neka je prvi Pericin korak dužine k stepenika. Tada on može da dovrši penjanje na B(n-k) načina. Kako dužina k prvog koraka može biti bilo koji broj od 1 do min(d,n), dobijamo sledeću relaciju:

$$B(m) = \begin{cases} 1 & za\ m = 0 \\ \sum_{k=1}^{\min(d,m)} B(m-k) & za\ m > 0 \end{cases}$$

Na osnovu ove formule, algoritam se jednostavno zapisuje:

```
input(d,n)
B(0) = 1;
for m = 1 to n
        s = 0;
        for k = 1 to min(d,m)
                s = s+B(m-k);
        B(m) = s;
output(B(n));
```

Optimalnost podstrukture ovog problema ogleda se u tome što broj načina da se pređe n stepenika koji počinju korakom od k stepenika mora biti jednak upravo broju načina da se pređe n-k stepenika. Dakle, rešenje problema sadrži u sebi rešenja podproblema, pa se može dobiti odgovarajućim kombinovanjem pogodno izabranih podproblema.

Podproblemi se očigledno preklapaju: na primer za n = 10 i d = 4, problem P(5) od m = 5 stepenika nam je potreban za rešavanje svakog od problema od P(6), P(7), P(8) i P(9). Drugim rečima, ova 4 problema se preklapaju jer imaju zajednički podproblem P(5) (i ne samo njega).

Napomena: kako ovo nije problem optimizacije (nego efikasnije izračunavanje vrednosti funkcije zadate rekurentno), termin „optimalnost podstrukture“ nije sasvim odgovarajući. U nedostatku boljeg, mi ćemo ipak koristiti ovaj termin u smislu kako je ranije objašnjeno.

Primer 3:

Voćara sa P pogona raspolaže sa N tona nafte i treba da preradi V tona voća. Data tabela

A opisuje moguće profite pogona u zavisnoti od raspoložive količine sirovina (voća) i energenata (nafte). Preciznije, za svako $1 \le p \le P$, $1 \le n \le N$, $1 \le v \le V$, broj A[p][n][v] predstavlja profit koji ostvaruje pogon p ako raspolaže sa n tona nafte i v tona voća (n i v su celi brojevi).

Raspodeliti raspoloživu količinu nafte i voća u celim tonama na P pogona tako da ukupan profit bude maksimalan, odnosno naći takve celobrojne n[p], v[p], p=1, ..., P, da je ∑n[p]=N, ∑v[p]=V, a da ∑A[p][n[p]][v[p]] bude maksimalna.

Rešenje:

Označimo sa B[p][n][v] maksimalan profit u prvih p pogona sa ukupno n tona nafte i v tona voća i neka je poslednji pogon pri optimalnoj raspodeli dobio k tona nafte i j tona voća. Tada i preostalih n-k tona nafte i v-j tona voća mora biti raspodeljeno na prethodnih P-1 pogona na optimalan način. Dakle, problem ima optimalnu podstrukturu.

Zbog toga, ako smo rešili problem za manji broj pogona pri svakoj količini energenata i sirovina, dovoljno je da pokušamo da novododatom pogonu dodelimo svaku moguću količinu energenata i sirovina i vidimo koja dodela u kombinaciji sa ranije pronađenom optimalnom raspodelom preostalih količina daje maksimalan profit. Preciznije, važe formule:

$$[p][0][v] = 0$$
$$[p][n][0] = 0$$
$$[1][n][v] = A[1][n][v],$$
$$[p][n][v] = \max_{\substack{0 \le k \le n \\ 0 \le j \le v}} \{B[p-1][n-k][v-j] + A[p][k][j]\}$$

Algoritam nalaženja optimalne raspodele i maksimalnog profita:

```
input(P, N, V, A) for
p = 1 to P
        for n = 0 to N
                A[p][n][0] = 0;
                B[p][n][0] = 0;
        for v = 0 to V
                A[p][0][v] = 0;
                B[p][0][v] = 0;
for n = 1 to N
        for v = 1 to V
                B[1][n][v] = A[1][n][v];
for p = 2 to P
        for n = 1 to N
                for v = 1 to V
                        B[p][n][v] = 0;
                        for k = 0 to n
                                for j = 0 to v
                                        if (B[p][n][v] < B[p-1][n-k][v-j] + A[p][k][j])
                                                B[p][n][v] = B[p-1][n-k][v-j] + A[p][k][j]
                                                BestK[p][n][v] = k;
                                                BestJ[p][n][v] = j;
OutputDistribution(BestK, BestJ, P, N, V)
output("Najveci profit je ", B[P][N][V]);
```

Brza rekurzivna funkcija za ispis tražene optimalne raspodele:

```
OutputDistribution(BestK, BestJ, p, n, v)
    if (P>0)
        OutputDistribution(BestK, BestJ, p-1, n-BestK[p][n][v], v-BestJ[p][n][v]);
        output(BestK[p][n][v], BestJ[p][n][v]);
```

Složenost algoritma je $O(P \cdot N^2 \cdot V^2)$.

Iz navedenih primera vidimo da tabela koja se popunjava pri rešavanju zadatka metodom dinamičkog programiranja može imati različit broj dimenzija (u ovim primerima od jedne do tri dimenzije), i da vreme izvršavanja algoritma može biti srazmerno različitim stepenima ulaznih veličina. Međutim, i pored ovih razlika kompletan postupak od analize do zapisa algoritma je suštinski isti u sva tri primera.